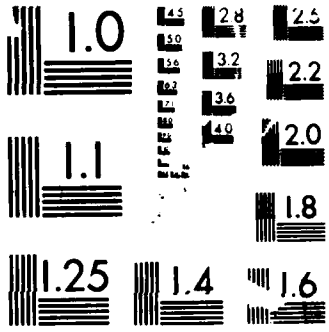
AD-A187 822 OPTIMAL PERFORMANCE OF DISTRIBUTED SIMULATION PROGRAMS 1/1
(U) UTAH UNIV SALT LAKE CITY DEPT OF COMPUTER SCIENCE
S M SHOPE ET AL. 1987 UUCS-87-023 N00014-87-K-0184
UNCLASSIFIED F/G 12/5 NL



(12)

OPTIMAL PERFORMANCE OF DISTRIBUTED SIMULATION PROGRAMS

Steven M. Swope and Richard M. Fujimoto
Computer Science Department
3190 Merrill Engineering Building
University of Utah, Salt Lake City, Utah 84112

Technical Report No. UUCS-87-023

1987

TECHNICAL REPORT

Department of
Computer Science

University of Utah
Salt Lake City, Utah

OPTIMAL PERFORMANCE OF DISTRIBUTED SIMULATION PROGRAMS

Steven M. Swope and Richard M. Fujimoto
Computer Science Department
3190 Merrill Engineering Building
University of Utah, Salt Lake City, Utah 84112

Technical Report No. UUCS-87-023

1987

N00014-87-K-0184

ABSTRACT

This paper describes a technique to analyze the potential speedup of distributed simulation programs. A distributed simulation strategy is proposed which minimizes execution time through the use of an *oracle* to control the simulation. Because the strategy relies on an oracle, it cannot be used for practical simulations. However the strategy facilitates performance evaluations of distributed simulation strategies by providing a useful point of comparison and can be used to determine the suitability of specific applications for implementation on a parallel computer. Based on the proposed strategy, a tool has been developed to determine the maximum performance which can be achieved from a distributed simulation program. In this paper we describe the technique and its use in evaluating the parallelism available in distributed simulators of parallel computer systems.

Keywords: discrete simulation, distributed simulation, parallel processing.

1. INTRODUCTION

Computer simulation is widely used to evaluate and design systems. However, due to the complex nature of many systems, detailed simulations often require too much time to be practical. For example, gate level simulations of large VLSI circuits can require *months* of computer time (Pfister1982, Franklin1984).

Distributed simulation, the execution of discrete event simulation programs on a multiple processor computer, is one possible solution to this problem. The emergence of parallel computers in the commercial marketplace has increased interest in this approach. Machines such as the BBN Butterfly™, the Intel iPSC™, and the NCube/Ten™ already provide users access to hundreds of powerful microprocessors in a single, integrated system.

However, distributed simulation has yet to be proven as a viable method for obtaining significant speedups of discrete event simulation programs. Empirical studies of distributed simulations of queuing networks have shown that some networks which initially appear to exhibit a reasonable degree of parallelism may in fact yield very disappointing performance (Reedappear). In some cases the distributed simulator executed more slowly than a single processor event list implementation.

When a distributed simulation program initially yields poor performance, the programmer is faced with a very difficult optimization problem. Poor performance may result from a number of causes. The simulation program may have been poorly partitioned into individual processes. Bottleneck processes may have been inadvertently created. Alternatively, the distributed simulation strategy may be the culprit. Any distributed simulation strategy will require some overhead to ensure that events are properly sequenced. This overhead may take the form of null messages, deadlock detection and recovery algorithms, or rollbacks (Chandy1979, Chandy1981, Jefferson1985). In general, determination of the ultimate cause of poor performance is a difficult, time consuming problem. Even uniprocessor simulation programs tend to be large, complex pieces of software with components that interact in subtle ways. A *distributed* implementation introduces complex dynamic parallel interactions, making the task even more difficult.

New tools are required to analyze distributed simulation programs in a convenient way in order to aid optimization efforts. In particular, a means of separating the behavior of the distributed simulation *strategy* from that of the simulation program is necessary. It is clear that some simulation programs are inherently sequential, and efforts to use distributed simulation techniques to improve their performance will be wasted. Such programs should be identified quickly to avoid futile attempts to obtain an efficient distributed implementation. On the other hand, application programs that exhibit a high degree of parallelism should also be quickly identified.

This paper will describe a tool which has been developed to measure the parallelism of a simulation program by measuring the speedup which can be obtained when an optimal distributed simulation strategy is used. Results from using this tool provide an upper bound on performance for a given distributed simulation program. This approach is similar in spirit to the VMIN algorithm which was developed to evaluate paging algorithms in virtual memory systems (Prieve1976). The optimal simulation strategy, called OSim, relies on an oracle to eliminate distributed simulation overheads such as null messages, deadlock detection and recovery, and rollback. Using OSim, a process blocks only when data dependencies dictate that it must wait for another message to arrive.

The analysis tool was developed in the context of the Simon simulation system, an object-oriented simulation package designed for functional and instruction level simulation of parallel computer systems

(Fujimoto1985, Swope1986). Simon currently uses a uniprocessor host, although it was designed to facilitate implementation on a parallel processor. The particular tool developed for this study was designed using Simon, however the technique which is used is applicable to any distributed simulation program.

We will first discuss the simulation model and the optimal simulation strategy, OSim. Then the tool which was developed using Simon will be described. Finally, parallelism measurements of several simulators for various parallel computer architectures will be presented.

2. THE DISTRIBUTED SIMULATION PROGRAM

We assume the system being modeled consists of some number of *physical processes* which interact in some manner. The simulation program consists of a collection of *logical processes* (LPs), each modeling a single physical process. For example, a multicomputer system might be divided into a collection of microcomputers and an interconnection switch, e.g. a global bus. The most natural mapping of this physical system to logical processes is to create one instance of a "switch" process to model the bus and a "processor" process to model each microcomputer.

Interactions between physical processes are modeled by timestamped messages passed between the corresponding logical processes. The timestamp indicates the simulated time at which the message arrives at the receiving process. Timestamps ensure that events are simulated in the proper sequence. Each LP must process messages in non-decreasing timestamp order. Logical processes repeatedly wait for the next message to be received, simulate the physical process, and then send zero or more message to other processes.

Ensuring that each LP processes messages in non-decreasing time stamp order is at the heart of the distributed simulation problem. Consider the situation depicted in figure 1 below. Assume that each process is executing on a different processor. LP A is ready to process another incoming message. Messages with timestamps of 10 and 50 are waiting to be processed. However, process A has yet to receive a message from process D. Since each process must act upon messages in non-decreasing timestamp order, process A is faced with a dilemma: should it wait for the next message from process D or should it go ahead and process the timestamp 10 message? If the timestamp 10 message is processed, then process A risks simulating events out of order. If it decides to wait, deadlock can result unless appropriate precautions are taken. Several solutions to this problem exist, each requiring a certain amount of overhead to ensure messages are properly sequenced (Misra1986, Jefferson1985).

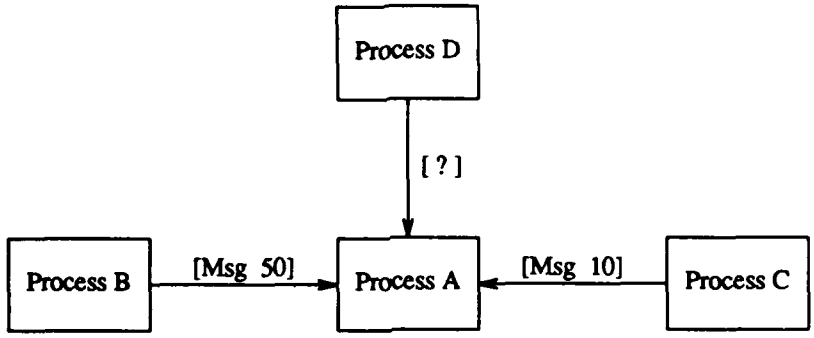


Figure 1: *Sample distributed simulator.*

3. OSIM: AN OPTIMAL SIMULATION STRATEGY

OSim is a distributed simulation strategy which eliminates the overhead associated with existing distributed simulation strategies. These existing strategies use mechanisms such as null messages, deadlock detection and recovery, and checkpoints and rollbacks to ensure correctness. OSim does not require such mechanisms because it relies on an oracle to direct logical processes to either proceed and process messages which have arrived or to wait. OSim is not a practical algorithm for use in distributed simulation implementations because the simulation program must first be executed to completion to generate the oracle.

Using OSim, the situation in figure 1 can be resolved, because process A can examine the oracle to determine if the next message from process D will have a timestamp smaller than 10. Blocking is therefore limited to those cases in which the next event that should be processed is a message that has not yet arrived. While OSim does not provide a practical solution to the distributed simulation problem, it does provide a means to measure the maximum amount of speedup which a distributed simulation program might obtain and thereby provide an upper bound on performance.

A key component of the optimal distributed simulation strategy is the oracle which is used to notify processes when they may proceed and when they must wait for one or more additional message(s). The oracle is created by first executing the simulation program to completion e.g., on a conventional *uniprocessor* using an event list to ensure events are properly sequenced. The strategy is analagous to the "instant replay" technique which has been proposed to debug parallel programs (LeBlanc1987).

The oracle contains a sequence of oracle elements, one for each event i.e., message, generated by the simulator. Each message is uniquely identified by a tag containing two fields: a unique number identifying the logical process which generated the message and a message sequence number to distinguish messages generated by that process. An

oracle entry consists of the destination logical process number and the message tag. During the uniprocessor simulation, entries are placed in the oracle as the messages are removed from the event queue and passed to the logical process. A timestamp ordering is therefore preserved for the messages received by each destination process.

The information in the oracle log is used by the *distributed* simulation program to identify the next message which should be delivered to each logical process. The following algorithm is executed when the logical process is ready to process another message:

```
NextMsgNumber := ConsultOracle( );
Found := SearchInputQueues(NextMsgNumber);
IF NOT Found THEN
        WaitForMsg(NextMsgNumber);
ENDIF;
Msg := RemoveFromInputQueue (NextMsgNumber);
RETURN Msg;
```

The ConsultOracle procedure returns the oracle entry for the next message which should be processed by the logical process calling it. SearchInputQueues examines the list of messages which have arrived, but have not yet been processed, and returns a boolean value of true if the message is currently available. If received messages are stored in timestamp order in a buffer associated with the input port on which the message arrived, only the first message of each buffer needs to be checked. If the message is found, it is removed from the input buffer and the logical process is allowed to simulate the activity associated with the message. Otherwise, the logical process blocks until the desired message arrives.

4. THE SIMON KERNEL

The OSim algorithm was implemented in the context of the Simon simulation system, so we will first make a brief digression to describe important features of Simon before describing the tool. Simon provides a flexible and adaptable framework for constructing simulators for a wide variety of parallel computer systems (Fujimoto1985). A simulator consists of a set of software building blocks. Each building block i.e., object, simulates a specific component of the parallel system. Objects may be defined in terms of other objects, supporting a hierarchical view of the system. Simon

provides a suite of procedures, data types and variables that define the interface to the programmer. Software in the Simon kernel consists of:

- a set of basic, low level primitives necessary to support an object-oriented simulation environment, and
- a set of library modules that use low level kernel primitives to build higher level mechanisms.

A simulation program in Simon includes a collection of autonomous objects. Interactions between objects are accomplished using timestamped messages. Each object defines a number of input and output ports through which all messages must pass.

The basic mechanisms provided by the Simon kernel include facilities for:

- instantiating objects,
- creating ports,
- interconnecting ports,
- exchanging timestamped messages, and
- specifying hierarchical structures.

4.1. SIMON OBJECTS

The main program is responsible for instantiating each object and connecting its ports. Each object is defined by an *object procedure* which begins execution as a coroutine when the object is instantiated. The object procedure first defines ports used by the object, and then simulates the behavior of the object. An object is equivalent to a logical process, described earlier.

4.2. SIMON PORTS

Interactions between an object and its external environment are through messages sent on output ports and received on input ports. No restrictions are placed on the number or type of ports an object can create. Although communicating objects must agree on the type and format of information transmitted through the ports, objects do not in general know with which, or even with how many other objects it is communicating. This increases the autonomy of each object and facilitates arbitrary interconnections.

Any output port may be connected to any input port regardless of how many other connections have already been established to either port or on which object the ports reside. Messages from several output ports may be merged by

connecting them to a single input port. Such messages are received in non-decreasing timestamp order. The order in which messages with the same timestamp are received is nondeterministic. Conversely, an output port may fanout to several input ports, implementing broadcast or multicast communications.

4.3. THE SIMON LIBRARY

Although application programmers may use the mechanisms defined in the kernel directly, most use higher level mechanisms built on top of these primitives, as provided in the Simon library. Some of the facilities include:

- queuing and priority queue abstractions
- buffered input ports
- arrays of ports
- random number generators
- explicit advancement of simulated time

The Simon kernel passes incoming messages to an object in non-decreasing timestamp order as they arrive, with no buffering. Two types of buffered input ports are provided - one which provides an unbounded first-in-first-out queue to hold arriving messages and another which provides semantics similar to shared memory. Queued input ports allow programs to wait for messages to arrive on specific input ports while messages arriving on other ports are automatically buffered. The second mechanism, referred to as registers, is associated with local memory variables within an object. The contents of incoming messages are automatically written into this memory when the message arrives, overwriting the previous contents. This is useful, for example, if the remote object is generating status information which another object reads.

5. PARALLELISM IN SIMULATION PROGRAMS

In this study, a uniprocessor simulation was used to study a distributed simulator executing a parallel application program. A uniprocessor based tool was developed to measure the speedup which can be obtained when the optimal distributed simulation strategy (OSim) is used. A distributed implementation of the tools is currently under development.

This study is similar in spirit to the work done by Livny (Livny1985). Measurements of parallelism in switch level simulations were also made by Frank (Frank1986). Unlike previous studies, the technique presented here may be easily extended for implementatior on a parallel processor. Also, highly accurate timing statistics are obtained because

detailed instruction level simulation of the distributed simulation program is used. However, parameterized lumped-sum costs are used to instrument operating system calls and communications overhead.

The distributed simulator is shown in figure 2 below. It consists of the Simon and OSim simulation kernels, the virtual machine layer, and the application layer. Support for basic simulation primitives is provided by the kernels. The virtual machine layer provides the user with an application oriented interface to the kernel primitives. The application layer consists of the user's application code.

The parallelism measurement program OSim, is implemented as a software module which is inserted between the virtual machine layer and the Simon kernel (see figure 2). It is completely transparent to the application, having an interface identical to that of the Simon kernel. Since OSim was built on top of the Simon kernel and uses many of Simon's facilities, it was relatively simple to implement. OSim required less than 1000 lines of code and took approximately one man-month to implement.

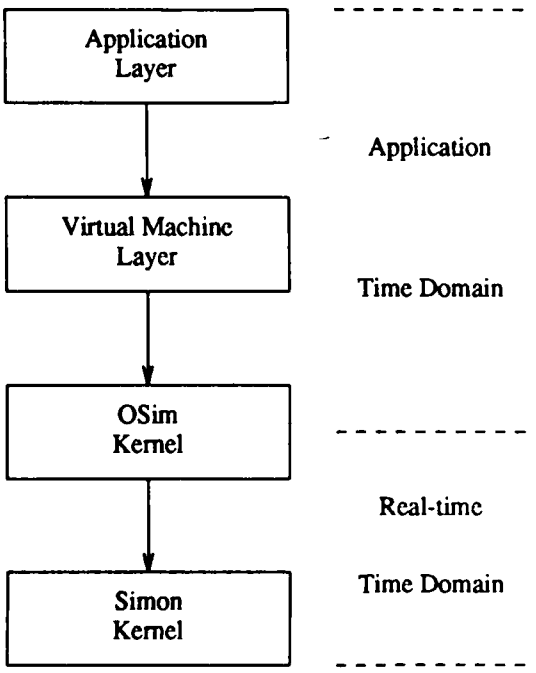


Figure 2: *Time Domains.*

5.1. TIME DOMAINS

In performing a simulation of a distributed simulator, two distinct time domains must be maintained (see figure 2):

(1) The application program deals with time in the *application domain*. Timestamps in this domain are called *application timestamps* and refer to time in the physical system being simulated.

(2) The *distributed simulator time domain* corresponds to time which elapses while the distributed simulator is executing. Timestamps in this domain are referred to as *real-time timestamps*, since they refer to real time in the frame of reference of the distributed simulator.

Both of these should be viewed as independent of each other, and independent of the passage of 'wall-clock' time as the distributed simulation executes on its uniprocessor host, in this case a VAX.

Messages generated by the application program have application timestamps which are meaningful in the application time domain. The OSim kernel encapsulates each message, including its application timestamp, into an OSim message. The timestamp of the encapsulated message is a real-time timestamp which indicates the time in the distributed simulation at which the message is received. The Simon kernel ensures that OSim messages are processed in non-decreasing real-time timestamp order, faithfully reproducing the behavior of the distributed simulator. When the message is received, the OSim kernel removes the message encapsulation and passes the original message to the receiving logical process.

Just as the Simon kernel ensures that real-time timestamps are processed in non-decreasing timestamp order, the OSim kernel ensures that application programs process incoming messages in non-decreasing *application* timestamp order. This is crucial to ensure correctness in the simulation of the distributed simulator, otherwise the simulation of the application program would be erroneous.

5.2. DIRECT-EXECUTION SIMULATION

Traditionally, instruction level simulation is implemented through a software interpreter which incurs a severe performance penalty of two to three orders of magnitude when compared to direct execution (Tamir1981). In this study the problem is overcome by directly executing the application program on the host processor rather than through a software interpreter (Fujimoto1983, Campbell1985).

First, the program is compiled to machine code for the *host* machine (a Vax™), and then instrumented. Basic blocks of code which may only be entered (exited) at the first (last) instruction of the block are identified, and an

increment instruction is inserted at the end of each block to indicate the amount of time required to execute this code on the target machine. This allows a relatively crude, but efficient timing model to be used based on the relative MIP rates of the host and target processors. The simulation programs used in this study rely heavily on the direct execution method to achieve efficient execution.

6. VIRTUAL MACHINE AND APPLICATION LAYERS

The virtual machine layer uses some subset of the mechanisms and abstractions provided by the kernel layers to provide an application specific interface to the application programmer. For example, one user interface might support development of application programs for an MIMD machine. Another might provide a hardware oriented interface designed for modeling switching components. Since both user interfaces are based on an integrated set of tools provided by the kernels, both of these different interfaces may be used in a single simulation program.

User interfaces have been developed for a number applications. In particular, several emulators of parallel architectures have been developed and used at the University of Utah. We shall briefly describe three interfaces here: an SIMD machine, a systolic array, and an asynchronous hypercube-based MIMD machine. Later, results will be presented which describe the parallelism available in each of these interfaces for typical application programs.

6.1. THE SIMD USER INTERFACE

The SIMD user interface models a machine consisting of 32 PEs, a control unit, and a data switch (see figure 3). Each PE contains a set of registers and local memory and has a simple "LOAD/STORE" architecture. The switch is capable of performing any permutation and can transmit a single word of data to and from each of the 32 PEs every machine clock cycle. The control unit is a general purpose processor whose instruction set consists of traditional sequential instructions which are executed within the control unit, and parallel instructions which are broadcast to the PEs for parallel execution.

Users of the SIMD machine develop application programs which are executed by the control unit. In the current implementation, sequential instructions are programmed using sequential Modula-2 code. Parallel broadcast instructions are implemented through procedure calls. A procedure is defined for each broadcast instruction supported by the SIMD architecture. Each broadcast instruction includes a mask which indicates which PEs are to be enabled and which are disabled during execution of the instruction. At the same time, a switch setting is also specified to control the configuration of the interconnection network.

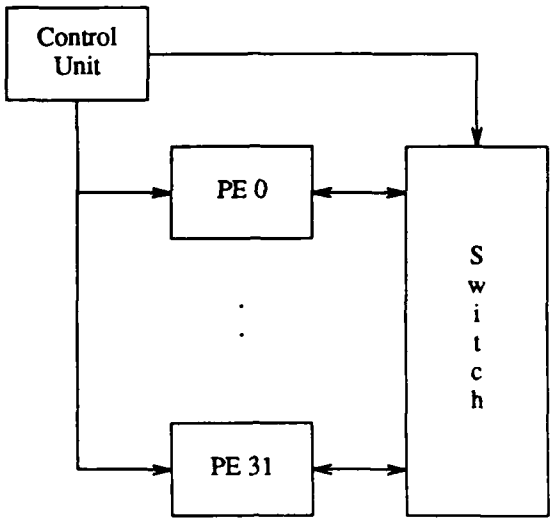


Figure 3: *SIMD Virtual Machine.*

6.2. THE SYSTOLIC ARRAY USER INTERFACE

The systolic array interface is based on a two-dimensional grid of processing elements. Users specify procedure(s) describing the behavior of PEs, and indicate which procedures are mapped to which grid points. The user interface automatically interconnects each PE with its neighboring PEs. Mechanisms are also provided to feed data into the array and to print results as they leave.

Each PE procedure defines input and output registers in any of the eight compass directions to indicate where incoming data is to be expected and where results are to be sent. Local variables are associated with input registers which are automatically updated when data arrives. The code for the procedure typically contains a loop which computes a new value from these local 'input' variables, stores the results into output registers, and then waits for the next clock cycle to begin. The latter function is accomplished by calling the "Wait4Clock" procedure defined by the user interface.

6.3. THE HYPERCUBE USER INTERFACE

The user of the MIMD hypercube interface must develop Modula-2 code for processors in the hypercube. In the current implementation, hypercubes containing up to 256 processors may be simulated. Interconnection of processing elements again is performed automatically by the user interface. Synchronous message passing primitives are provided which allow the program running in each node to communicate with neighboring nodes.

6.4. APPLICATIONS

The first application performs a bubble sort using the SIMD simulator. Each PE is provided with two data elements. The elements are compared and the larger value is passed through the data switch to the PE above it. The resulting two elements are compared and the smaller one is passed back to the PE below. This continues until the data elements are completely sorted.

The second application performs a matrix multiplication on a stream of vectors using the systolic array simulator. Each of the PEs is preloaded with an element from a 5 x 5 matrix. The stream of vectors is skewed and fed into the top of the array, and flows through to the bottom. The sums of products are accumulated and passed from left to right with the final results emerging from the right edge of the array.

The third application performs a simple image processing algorithm on the hypercube simulator. Each PE is preloaded with an equal-sized square section of a 96 x 96 pixel image. Each pixel is averaged with its four neighboring pixels in the north, south, east, and west directions. Thus, communications are required only between neighboring PEs.

7. PARALLELISM MEASUREMENTS

The above applications were each simulated using the OSim distributed simulation environment. Each exhibited a different set of execution characteristics. The applications incorporated the user interfaces from the three parallel architectures described above. The sections which follow present some initial results in using OSim to predict the maximum performance which can be achieved by a distributed simulator.

7.1. HARDWARE MODEL OF THE SIMULATOR

The hardware on which the distributed simulation program is executed is assumed to be a general purpose, MIMD machine. Each processor is assumed to have a VAX like architecture. We assume that a processor can communicate with any other processor through a dedicated communications channel of some fixed bandwidth. It is assumed that communications are reliable.

To expose all parallelism available in the application program, it is assumed that each object is mapped to a different processor. The OSim kernel determines the execution times of application programs based on the following costs:

(1) Each machine instruction requires 1.0 microseconds of execution time. Application programs are compiled using the DEC Modula-2 compiler for the VAX.

(2) System calls (such as memory allocation) require 100.0 microseconds

(3) Each interprocessor communication requires 100.0 microseconds to execute a message send or receive primitive, plus the time required to 'physically' transmit the message over the communications channel.

(4) The bandwidth of the communications channel is assumed to be 10 Mbits/sec.

7.2. RESULTS

The direct execution technique was used to measure the execution times of each application for the uniprocessor version as well as the distributed version of the program. Speedup figures were computed and reported in Table 1 below.

Table 1: Potential Speedup of Distributed Simulators

Application	Processors	Speed-Up	% of Ideal
32 PE SIMD	34	23.4	68.8
Systolic Array-1	28	7.2	25.7
Systolic Array-2	41	30.5	74.4
Hyper-Image-4	4	3.87	96.8
Hyper-Image-16	16	15.0	93.4
Hyper-Image-64	64	54.9	85.7
Hyper-Image-256	256	198.4	77.5

The SIMD application had a configuration consisting of 32 PE objects, 1 data switch object, and 1 control unit object. Using a total of 34 processors, a speed-up of 23.4 was obtained, 68.8% of ideal. It was noticed that there was a serial execution pattern between the PEs, the switch, and the control unit, which had a limiting effect on the amount of parallelism in the execution.

The systolic array application was initially configured with 25 PE objects, 2 I/O objects (one to feed data into the array and one to remove results), and a clock object. Using 28 processors, a speed-up of 7.2 was obtained, which is only 25.7% of ideal. The bottleneck in this application was the large granularity of the I/O objects. The configuration was reorganized into 25 PE objects, 15 I/O objects (10 to feed individual input streams and 5 to remove individual output streams), and 1 clock object. This new configuration used 41 processors, and a potential speed-up of 30.5 was obtained which is 74.4% of ideal.

The hypercube application used square configurations of 4, 16, 64, and 256 PE objects. The results were particularly encouraging, yielding potential speed-ups as high as 97% of ideal. However, it should be noted that this percentage drops significantly as the granularity of the computation (pixels per processor) is reduced. This is because communica-

tions overhead becomes more significant.

8. CONCLUSIONS AND FUTURE RESEARCH

The OSim kernel, a tool for measuring the speedup of distributed simulation programs using an "optimal" simulation strategy, was used to measure the potential speedup available in simulators for several parallel architectures. Potential speedup measurements were reported and the initial results for simulating parallel computer architectures on a distributed simulator are encouraging. An overview of the Simon simulation system was presented, along with several user interfaces for different parallel architectures.

Work will continue in evaluating the relationship between the speedup of a distributed simulator and characteristics of the system being simulated. A distributed version of Simon using the BBN Butterfly Multiprocessor™ is currently under development in order to compare actual speedup figures using real distributed simulation mechanisms with those predicted by the optimal simulation policy. Finally, work is also in progress to determine strategies for partitioning simulation programs into objects and mapping these objects to general purpose multiprocessor hardware.

ACKNOWLEDGEMENTS

This work was supported in part by IBM under a faculty development grant and from ONR contract number N00014-87-K-0184.

REFERENCES

Campbell, W. B., "The Efficient Modeling of Processor Behavior and Performance," Master's Thesis, University of Utah, Salt Lake City, Utah (September 1985).

Chandy, K. M. and Misra, J., "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering* SE-5(5) pp. 440-452 (Sept. 1979).

Chandy, K. M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM* 24(4) pp. 198-206 (April 1981).

Frank, E., "Exploiting Parallelism in a Switch-Level Simulation Machine," *Proc. 23rd Design Automation Conference*, pp. 209-215 (July 1986).

Franklin, M. A., Wann, D. F., and Wong, K. F., "Parallel Machines and Algorithms for Discrete-Event Simulation," *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 449-458 (August 1984).

Fujimoto, R. M., "Simon: A Simulator of Multicomputer Networks," Electronics Research Laboratory Report No. UCB/CSD 83/137, University of California, Berkeley, CA (1983).

Fujimoto, R. M., "The Simon Simulation and Development System," *Proceedings of the 1985 Summer Computer Simulation Conference, Chicago, Illinois* 17(1) pp. 123-128 (July 1985).

Jefferson, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems* 7(3) pp. 404-425 (July 1985).

LeBlanc, T. J. and Mellor-Crummey, J. M., "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers* C-36(4) pp. 471-482 (April 1987).

Livny, M., "A Study of Parallelism in Distributed Simulation," *Proceedings of the SCS Distributed Simulation Conference San Diego, California* 15(2) pp. 94-98 (January 1985).

Misra, J., "Distributed-Discrete Event Simulation," *ACM Computing Surveys* 18(1) pp. 39-65 (March 1986).

Pfister, G. F., "The Yorktown Simulation Engine: Introduction," *Proc. 19th Design Automation Conference*, pp. 51-54 (June 1982).

Prieve, B. G. and Fabry, R. S., "VMIN - An Optimal Variable Space Page Replacement Algorithm," *Communications of the ACM* 19(5) pp. 295-297 (May 1976).

Reed, D. A., Malony, A. D., and McCredie, B. D., "Parallel Discrete Event Simulation: A Shared Memory Approach," *IEEE Transactions on Software Engineering*, (to appear).

Swope, S. M. and Fujimoto, R. M., "Simon II Kernel Reference Manual," Technical Report UUCS-86-001, University of Utah, Salt Lake City, Utah (1986).

Tamir, Y., "Simulation and Performance Evaluation of the RISC Architecture," Electronics Research Laboratory Memorandom No. UCB/ERL M81/17, University of California, Berkeley, CA (March 1981).

END

FILMED

FEB. 1988

DTIC